Incremental auto-tuningを用いたOpenCLのhybrid並列化

著者	若谷 彰良
雑誌名	甲南大学紀要. 知能情報学編
巻	13
号	2
ページ	103-110
発行年	2021-02-10
URL	http://doi.org/10.14990/00003694

論　文

Incremental auto-tuningを用いたOpenCLのhybrid並列化

若谷彰良

甲南大学 知能情報学部
神戸市東灘区岡本８－９－１, 658-8501

(受理日 2020 年 11 月 16 日)

概 要

最近のプロセッサにはGPUと複数のプロセシングコアの両方を内蔵するものがあり, GPUのGPGPUに基づく並列処理とCPUにおけるマルチスレディングに基づく並列処理を同時に行うhybrid並列が利用可能になっているが, アプリケーションによってCPUとGPUの性能の比は異なり, 最適な負荷分散をあらかじめ決定することは難しい. 著者らは, 以前に, 最適な負荷分散を実行時に決定する, on-the-flyな自動チューニング法を提案したが, 予備実行に用いる計算範囲の割合(*AutoRatio*)を事前に決める必要があった. そこで, 予備実行の割合を決定することなく, 実行時にincrementalに予備実行を行なう自動チューニング方法を提案し, 4種類のアプリケーションに対してその手法の有効性を確認した.

キーワード: ヘテロジニアス，マルチコア，負荷分散

1　はじめに

性能向上を図るために, システムを構成するパラメータを調整していくことをチューニングといい, その過程を自動的に行うことを自動チューニングという. 大規模な計算を行う数値計算では, 高速な計算機を利用するとともに, チューニングが重要な技法の一つである.

一方, 半導体の製造技術および設計技術の伸展とともに一つのチップ上に搭載されるトランジスタの集積度が高まり, プロセッサには多くのプロセシングコア（コア）が実装されるようになった. プロセシングコアの増加とともに, 周辺機器の一体化も進み, 例えば, Westmereアーキテクチャ以降のIntelのプロセッサには, GPUをも同一チップ上に搭載できるようになっている. GPUが搭載されている場合は, CPUとともに, GPUでも数値計算のような汎用計算を行うことで, 一層の高速化ができる可能性がある. GPUで汎用な計算を行うことをGPGPU (General Purpose computing on Graphics Processing Units)と呼ぶ. そこで, GPUでのGPGPUだけでなく, FPGAのプログラミングやメニーコアプロセッサなど, 様々な並列処理システムに対する並列プログラミングのフレームワークとして, OpenCLの利用が進みつつある [1]. さらに, 2.0以降のバージョンのOpenCLにおいては, SVM (Shared Virtual Memory)の利用ができるようになっており, GPUとCPUがメモリ空間を共有するので, データ移動のコストを考慮せずに, GPUとCPU間での負荷の分散を行うことができる.

前述の通り, GPGPU システムにおける性能向上のためのチューニング技法は重要であるが, OpenCL を用いた自動チューニングについてはこれまでに多くの研究が行われている [2], [3]. また, CPU と GPU のように, 異種のアーキテクチャを用いてアプリケーションの並列化を図ることを, heterogeneous 並列化もしくは hybrid 並列化と呼ぶが, 使用される計算方式が異なるので, 異種のアーキテクチャ間で負荷分散を最適に行うことは難しい. また, その最適化を自動的に行うことを自動チューニング手法と呼ぶ. 文献 [4] において, CPU (Intel Xeon プロセッサ) と GPU (Nvidia Tesla アーキテクチャ) から構成される GPGPU システムに対する, CPU と GPU の BLAS ルーチンの hybrid 並列の自動チューニングの手法について述べられている. キーパラメータを抽出し, BLAS ルーチンの高速化は可能であるが, キーパラメータを事前に抽出することが必要なので, そのためにオーバーヘッドが生じる. また, 文献 [5] においては, 一定の割合 (*AutoRatio*) のデータ部分に対する事前の予備実行を行ない, CPU と GPU の実行性能比を求め, その後, その性能比から計算された負荷分散を用いて hybrid 並列を行うことにより, 自動チューニングを実現している. しかし, 初めてアプリケーションを実行する場合など, 事前に *AutoRatio* を決定できれないケースではこの手法は用いることはできない.

本稿では, 事前に予備実行のサイズを決めず, 小規模な予備実行を実行時に繰り返し, 計測時間が安定した時点での平均実行時間で負散を決定するチューニング方法 (incremental 自動チューニング) を提案する. また, GPU が内蔵された Coffeelake アーキテクチャの Intel プロセッサを対象とし, 文献 [5] と同様に, 4 種類のアプリケーション, すなわち, スカラ変数の 2 個のアプリケーションと大規模配列の 2 個アプリケーションに対して, CPU と GPU の性能比を incremental 自動チューニングで求め, その評価を行う.

2 GPGPU システム

2.1 GPU 内蔵プロセッサ

本稿の実験においても, 文献 [5] と同様に, 表 1 に示す 2 台のコンピュータを使用する. CPU はいずれも Intel 製の Coffeelake アーキテクチャによるプロセッサで, GPU として UHD 630 を内蔵している. PC1 と PC2 のプロセシングコア数は 6 と 4 で, PC1 のプロセッサはハイパースレディングが利用可能なので, 12 スレッドまでの並行実行ができる. UHD 630 は 24 個の実行ユニット (192 個の演算ユニット) をもつ.

表 1: Specifications of PCs

	CPU		GPU	Mem.	OS
	no. of cores	no. of threads			
PC1	intel i7-8700 (3.2 GHz)		UHD 630	8 GB	Windows 10 Pro.
	6 cores	12 threads			
PC2	intel i3-8100 (3.6 GHz)				
	4 cores	4 threads			

また, OpenCL の SVM には 3 つ形態, 1) Coarse grain buffer, 2) Fine grain buffer, 3) Fine grain system があるが, PC1 及び PC2 のプロセッサ対応の OpenCL では, SVM の Fine grain buffer が利用可能であり, これによりメモリ間のデータ移動は無いので, CPU と GPU の負荷分散だけを考慮して, 最適な hybrid 並列が可能になる.

2.2 評価アプリケーション

本稿の実験では, 文献 [5] と同様に, 4 種類のアプリケーション, すなわち, 2 種類のスカラ変数のアプリケーションと 2 種類の大規模配列のアプリケーションを典型的なアプリケーションと考える.

大規模で均質なスカラ計算の例として, arctan 関数の微分を積分することで円周率を求める計算を行う. $[0,1)$ の区間を 256×10^6 及び 10^{10} の区間に分割して積分する計算で評価する. 2 個目のスカラ計算の例として, 8×10^6 未満の整数, 及び 16×10^6 未満の中の素数の数をカウントする計算で評価する.

一方, 大規模配列を用いる計算の例として, 行列積と行列ベクトル積を行う. 行列ベクトル積の実験では, 8192×8192 の行列と 8192 のベクトルとの積, 及び 16384×16384 の行列と 16384 のベクトルとの積で評価する. また, 行列積の実験では, 4096×4096 の行列間, 及びの 8192×8192 の行列間の積で評価する.

3 On-the-fly 自動チューニング

3.1 概要

本節では, 文献 [5] に記述した on-the-fly 自動チューニングの概要と, 前述した 4 個のアプリケーションに対する, Hybrid 並列の効果を実験結果をまとめる.

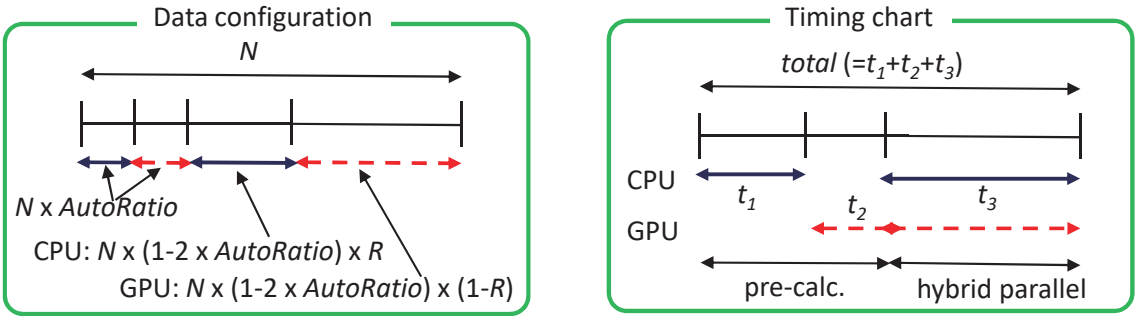

図 1: On-the-fly auto-tuning

PC においては, CPU が実行する部分の for ループの並列化には OpenMP を用い, スレッド数は, ハイパースレディングを用いた場合のハードウエアとして最大のスレッド数, すなわち 12 スレッドとする. GPU が実行する部分の並列化には OpenCL を用い, SVM として Fine grain buffer を利用し, GPU と CPU とで unified memory を共有する.

on-the-fly 自動チューニングでは, アプリケーションのサイズ N の一部を CPU 及び GPU で別々に実行し, 計測された実行時間を用いて, 最適な R の値を決定する.

図 1 に示すように, 達成されるスピードアップの値の最も高い部分から予備実行を行う部分の割合, すなわち *AutoRatio* を決める. つまり, データサイズ $N \times AutoRatio$ の部分を CPU および GPU で実行した時間を t_1 と t_2 とし, ここから残りのデータ部分を CPU と GPU が同じ実行時間 $(= t_3)$ で実行できる割合 (R 値) を, $R = t_2/(t_1 + t_2)$ で決め, CPU が $N \times (1 - 2 \times AutoRatio) \times R$ のデータ部分を, CPU が $N \times (1 - 2 \times AutoRatio) \times (1 - R)$ のデータ部分を, 同時に, hybrid 並列で実行する. 以上により, $t_1 + t_2 + t_3$ の合計の実行時間で全体の計算を行うことになる.

しかし, *AutoRatio* が小さいと, CPU と GPU の正確な性能比率が求められないので, 結果として最適な R を求めることはできないことになり, また, *AutoRatio* が大きい場合は $N \times (1 - 2 \times AutoRatio)$ が小さくなるので, 最適な hybrid 並列の効果を得にくくなる. そこで, 実験を用いてアプリケーション毎に最適な *AutoRatio* を求めることになる.

3.2 *AutoRatio* の決定と評価

表 2 に *AutoRatio* を 0.01 から 0.16 まで変化させた際のスピードアップを示す. 下線分が最適値である. すなわち, 与えられた *AutoRatio* の部分を予備的に実行して実行時間を計測し, それで決まった R の値で, 残りのデータ部分を hybrid 並列した場合の実行時間によりスピードアップを計算している.

表 2: Auto tuning on i7-8700 (PC1)

	0.01	0.02	0.04	0.08	0.16
pi	0.67	1.03	<u>1.1</u>	0.93	0.72
prime	1.7	1.96	1.68	<u>2.84</u>	1.9
matvec	1.05	1.05	1.14	<u>1.15</u>	1.13
matmul	1.116	1.34	1.53	1.68	<u>1.91</u>

PC1 で求めた *AutoRatio* の値を用いて, PC2 における hybrid 並列が最適になっているかを確認する. PC2 に on-the-fly 自動チューニングを適用した結果を表 3 に示す.

表 3: Results of auto-tuning on i3-8100 (PC2)

	pi		prime		matvec		matmul	
	R	speed-up	R	speed-up	R	speed-up	R	speed-up
Optimal	0.75	1.47	0.30	2.89	0.65	2.26	0.15	5.1
Auto-tuning	0.75	1.30	0.21	2.47	0.61	1.88	0.22	3.84

行列ベクトル積 (matvec) では, $AutoRatio = 0.08$ の予備実行で, $R = 0.61$ が求まり, その結果, スピードアップ 1.88 を達成し, 行列積 (matmul) では, $AutoRatio = 0.16$ の予備実行で, $R = 0.22$ が求まり, その結

果, スピードアップ 3.84 を達成した. また, 円周率計算 (pi) では, $AutoRatio = 0.04$ の予備実行で $R = 0.75$ が求まり, その結果, スピードアップ 1.30 を達成し, 素数カウント (prime) では, $AutoRatio = 0.08$ の予備実行で, $R = 0.21$ が求まり, その結果, スピードアップ 2.47 を達成した.

素数カウントにおいては, 最適の R の値との自動チューニングで得られた値との差がやや大きいが, 一定レベルのスピードアップは達成されていると考えられる. また, それ以外のアプリケーションに関しては, PC1 で決定した $AutoRatio$ の値による on-the-fly 自動チューニングにより, ほぼ最適な R の値が求まっていることが分かる.

4 Incremental 自動チューニング

4.1 概要

これまでに述べたように, あらかじめ, 予備実行を行う割合 ($AutoRatio$) を決定し, それに基づいて CPU と GPU の負荷分散, すなわち, R 値を決めることによる on-the-fly 自動チューニングは一定の性能を示すことができる. しかし, $AutoRatio$ をあらかじめ決められない場合, 例えば, 初めて実行するプログラムであれば, 予備的な評価がされていないので, $AutoRatio$ の値は不定である.

そこで, 小規模の予備実行を, 計測した実行時間の平均値が安定するまで数回繰り返し, それにより R の値を予想する Incremental 自動チューニングを第 2 の方法として提案する.

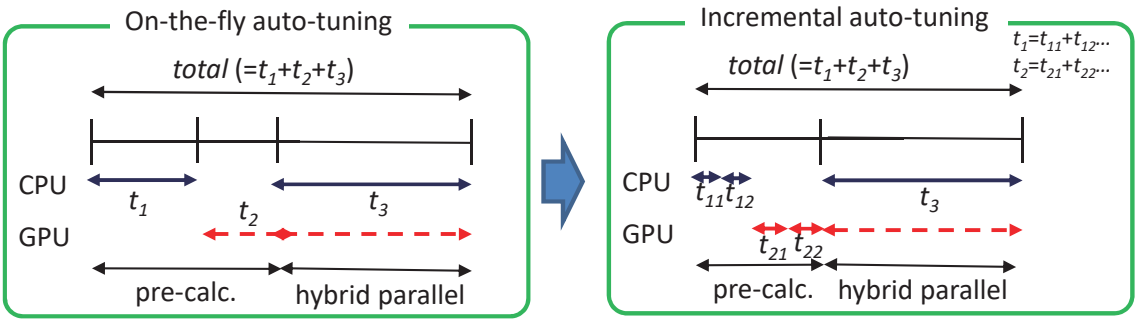

図 2: Incremental auto-tuning

図 2 に incremental 自動チューニングの概要を示す. $AutoRatio$ を用いた on-the-fly 自動チューニングであれば, 予備実行の時間が固定のオーバーヘッド時間となっていた. すなわち, t_1 が CPU の予備実行時間で, t_2 が GPU の予備実行時間である. そこで, incremental 自動チューニングでは, 計算全体の 0.5%の部分を 1 単位として, 実測を繰り返し, 実行時間の平均の差が 10%未満になった時点で予備実行を停止する. まずは, CPU で予備実行を行ない, 次に GPU で予備実行を行う. すなわち, CPU で 1 単位の予備実行を t_{1i} $(i \geq 1)$ の実行時間で行い, $\left| \frac{\sum_{i=1}^{n} t_{1i}}{n} - \frac{\sum_{i=1}^{n-1} t_{1i}}{n-1} \right|$ が一定の値以下になるまで繰り返す. さらに, GPU で 1 単位の予備実行を t_{2i} $(i \geq 1)$ の実行時間で行い, $\left| \frac{\sum_{i=1}^{n} t_{2i}}{n} - \frac{\sum_{i=1}^{n-1} t_{2i}}{n-1} \right|$ が一定の値以下

になるまで繰り返し, $\frac{\sum_{i=1}^{n_1} t_{1i}}{n_1}$ と $\frac{\sum_{i=1}^{n_2} t_{2i}}{n_2}$ の比率で R の値を決定する.

なお, GPU の予備実行に関しては, GPU の起動時間やメモリアクセスの効率性の理由で, 計算の種類およびデータのサイズによれば, GPU の予備実行が CPU よりもかなり遅い場合がある. そこで, GPU の予備実行では, 1 単位の実測値が CPU の 1 単位よりも 5 倍以上遅い場合は, 予備実行を繰り返さずに, R の値を 0.95 で固定にし, GPU の予備実行のオーバーヘッドを削減することにする.

4.2 評価

表 4 に, 素数探索において, 探索する範囲が 8×10^6 と 16×10^6 の場合の, hybrid 並列での CPU の実行割合である R 値 ($CPU : GPU = R : 1 - R$) と, CPU のみの実行時間に対するスピードアップを示す. opt は最適な R 値に基づく hybrid 並列の結果を, auto は固定サイズの予備実行を用いた on-the-fly 自動チューニングの結果を, inc は incremental 自動チューニングの結果を表す.

表 4: Results of prime with incremental auto-tuning

		size$= 8 \times 10^6$		size$= 16 \times 10^6$	
		R	speedup	R	speedup
PC1	opt	0.35	2.22	0.35	1.91
	auto	0.07	1.33	0.06	1.15
	inc	0.52	1.39	0.25	1.53
PC2	opt	0.30	2.89	0.35	2.94
	auto	0.21	2.47	0.17	2.31
	inc	0.45	2.13	0.33	2.79

一般的には, データサイズが大きい場合は, 予備実行もサイズを大きくとれるので, より精度高く, R 値を予測できると考えられる. PC1 では, on-the-fly 自動チューニングよりも incremental 自動チューニングの方が最適値に近いスピードアップを, 8 M ($= 8 \times 10^6$) のサイズよりも 16 M ($= 16 \times 10^6$) サイズの方が最適値に近いスピードアップとなっている. また, 16 M サイズでは, incremental 自動チューニングの R 値 ($= 0.25$) も, 最適値 ($= 0.35$) に近い値になっている. 一方, PC2 においては, 8 M サイズにおいては, on-the-fly 自動チューニングのスピードアップが, incremental 自動チューニングのスピードアップより優っているが, 16 M サイズでは, それは逆転し, incremental 自動チューニングの R 値 ($= 0.33$) は最適値 ($= 0.35$) に近い値になっており, スピードアップも最適値に近い. 素数探索に関しては, 固定の *AutoRatio* 値による on-the-fly 自動チューニング方式よりも, 可変的なサイズの予備実行を行う incremental 自動チューニング方式の方が優れている.

表 5 に, PC1 における各種アプリケーションに対する最適値 (opt), on-the-fly 自動チューニング (auto), incremental 自動チューニング (inc) の結果を示す. それぞれのアプリケーションにおいて, データサイズの小さい時と大きい時の 2 パターンの結果を示している. 前述の通り, 素数探索 (prime) は, incremental 自動チューニングが優れている. 円周率計算 (pi) では, incremental 自動チューニングの R 値は最適値とほぼ同じになっている. しかし, スピードアップが最適値よりもやや落ちているのは, 予備実行の部

分がオーバーヘッドとなっているからであり, これは, 他のアプリケーションでも同様の結果になる. また, on-the-fly 自動チューニングとの比較では, スピードアップでは劣るが, R 値の性能は高い.

行列積 (matmul) や行列ベクトル積 (matvec) では, on-the-fly 自動チューニングと比較した場合の incremental 自動チューニングの性能はやや劣る. 例えば, 8 K ($= 8 \times 10^3$) サイズの行列ベクトル積の場合, R 値に関しては on-the-fly 自動チューニング ($= 0.77$) が incremental 自動チューニング ($= 0.9$) よりも最適値 ($= 0.80$) に近く, スピードアップでも on-the-fly 自動チューニングが上回っている. しかし, 概して, incremental 自動チューニングで得られた R 値は最適値付近の値となっている. 例えば, 4 K ($= 4 \times 10^3$) サイズの行列積の場合, 最適な R 値が 0.55 であるのに対し, incremental 自動チューニングで得られた R 値は 0.58 である. よって, 事前に *AutoRatio* の値が分かっていない場合であれば, incremental 自動チューニング法がその代替えとして用いることが可能であると言える.

表 5: Results of incremental auto-tuning on PC1

		opt		auto		inc	
	size	R	speedup	R	speedup	R	speedup
pi	256×10^6	0.95	1.18	0.25	1.15	0.95	1.13
	10×10^9	0.90	1.24	0.89	1.14	0.95	0.79
prime	8×10^6	0.35	2.22	0.07	1.33	0.52	1.39
	16×10^6	0.35	1.91	0.06	1.15	0.25	1.53
matvec	8×10^3	0.80	1.78	0.77	1.22	0.95	0.99
	16×10^3	0.80	1.54	0.67	1.13	0.95	1.13
matmul	4×10^3	0.55	1.43	0.45	1.44	0.58	1.06
	8×10^3	0.60	1.21	0.51	1.25	0.46	0.98

なお, 以上の結果は PC2 でもほぼ同様であり, 一定レベルの incremental 自動チューニングの有効性は確認できる. 事前に *AutoRatio* を決定できる場合において, *AutoRatio* を用いた on-the-fly 自動チューニングとの使い分けをどのように行うかが今後の課題となる.

5 おわりに

近年のプロセッサには複数のプロセシングコア以外に GPU を内蔵するものがあり, CPU の並列処理と GPU における並列処理を同時に行う hybrid 並列が利用可能であるが, アプリケーション毎に CPU と GPU の性能比は異なり, 最適な負荷分散を事前に決定することは難しい. 本稿では, 予備実行の割合を決定することなく, 実行時に incremental に予備実行を行なう自動チューニング方法を提案し, 4 個のアプリケーションに対してその手法の有効性を確認した.

今後は, 異なるアーキテクチャやプロセッサを用いたのシステムでの評価を行い, incremental 自動チューニングの有効性を確認するとともに, on-the-fly 自動チューニングとの使い分けについての手法を確立していきたい.

謝辞

本研究の一部はJSPS科学研究費(基盤研究(C) 18K02920 (2018-2020), 基盤研究(C) 19K03018 (2019-2011))及び私立大学等経常費補助金特別補助「大学間連携等による共同研究」による.

参考文献

[1] https://www.khronos.org/opencl/ (As of 2019/8/17).

[2] C. Cummins, P. Petoumenos, M. Steuwer and H. Leather, "Autotuning OpenCL workgroup size for stencil patterns," in *Proc. 6th International Workshop on Adaptive Self-tuning Computing Systems*, CD-ROM, 8 pages, 2016.

[3] J. D. Garvey and T. S. Abdelrahman, "A strategy for automatic performance tuning of stencil computations on GPUs," *Scientific Programming*, vol. 2018, Article ID 6093054, 24 pages, https://doi.org/10.1155/2018/6093054, 2018.

[4] G. Bernabe, J. Cuenca, L. Garca and D. Gimenez, "Tuning basic linear algebra routines for hybrid CPU+GPU platforms," in *Proc. 14th International Conference on Computational Science*, pp. 30-39, 2014.

[5] A. Wakatani, "Evaluation of on-the-fly auto-tuning of hybrid parallelization on processors with integrated graphics," in *Proc. 4th South-East Europe Design Automation*, *Computer Engineering*, *Computer Networks and Social Media Conference*, CD-ROM, 5 pages, 2019.